Systems of Systems: Scaling Up the Development Process

Watts Humphrey

August 2006

TECHNICAL REPORT
CMU/SEI-2006-TR-017
ESC-TR-2006-017

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Systems of Systems: Scaling Up the Development Process

CMU/SEI-2006-TR-017
ESC-TR-2006-017

Watts Humphrey

August 2006

Software Engineering Process Management Program

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgements

The development of systems of systems (SoS) differs in many fundamental ways from that of traditional systems. This holds true for SoS-like systems, which have many but not all of the properties of an SoS. A principal objective of this report has been to identify those areas where these systems can and must take advantage of what has already been learned about systems development and systems management. Based on this work, and because of the willingness of many people to help with my education on SoS-like systems, I am now convinced that the large-systems lessons of the past can and must be applied to such future work and that, if they are not, we will experience many costly and possibly even disastrous program failures.

This report has been based on the work of a great many people, and I am grateful for their help and guidance. In particular, I appreciate the help of David Fisher, Suzanne Garcia, John Goodenough, Patricia Oberndorf, Dennis Smith, and Hal Wilson in clarifying these issues and in helping me to understand the fundamental differences between traditional large-scale systems development and the processes required to produce SoS-like systems. I did not appreciate this difference until Suzanne Garcia pointed out that it is like the difference between designing a building and planning a city.

In writing this report, many people have also contributed their time for reviewing document drafts and otherwise helping me with this work. For their help, I particularly thank David Carrington, Claire Dixon, Suzanne Garcia, Marsha Pomeroy-Huff, Bill Nichols, Patricia Oberndorf, Bill Peterson, and Hal Wilson.

I also very much appreciate all the work and insight provided by the prior work of the members of SEI's Dynamic Systems, Product Line Systems, and Software Engineering Process Management Programs. In addition, I thank Claire Dixon for her help in turning my early drafts into a finished technical report.

Executive Summary

Large and complex computer-based systems are now critical to the economic and military welfare of the United States and to much of the industrialized world. These systems form the backbone of modern military, business, and governmental operations, and without their continued support, our societies would be severely inconvenienced and even threatened. Unfortunately, the development of such systems has been troubled, and the systems needed in the future will be vastly more complex and challenging. If history is any guide, attempting to develop these future systems with the outmoded methods of the past will almost certainly yield unsatisfactory results.

The systems-of-systems (SoS) research area is extensive and involves a host of technical and management issues. Several groups, particularly the Dynamic Systems Impact Program and the Product Line Systems Program at the Software Engineering Institute, are addressing the technical issues of such systems. However, there has not yet been a concerted effort to define and understand the process management and control issues involved in the development, evolution, and operation of these systems. Since a number of groups are currently defining and starting to develop such systems, it is important to understand the suitability of the traditional development processes that will have to be used if newer and more appropriate methods are not defined and adopted.

While SoS design and development involves many technical, business, and management topics, this paper focuses only on the process-related aspects of building or evolving such very large systems. In summary, this report addresses the following question: In developing a large-scale software-intensive system of systems that is to have the properties generally characteristic of such systems, what processes, methods, and practices should be used?

An additional report objective is to outline those process and project-management areas where further research and development is needed. Such work will provide the foundation for extending known and proven best practices to guide organizations and groups in working with large-scale SoS-like systems. Note that an SoS-like system is one that has many but not all of the properties of an SoS.

This report reviews the problems of developing large-scale SoS-like systems and outlines steps for addressing them. The report has eight sections.

1. Future Large-Scale Systems
2. The Design of Large-Scale Systems
3. Critical Systems-Development Problems
4. The Scale-Up Problem

5. Quality Principles
6. Future Process Needs
7. SoS Process Strategy
8. Large-Scale Single-System Process Considerations

The first section, on the nature of future systems, summarizes current thinking about complex systems. For example, the Internet is a system-of-systems (SoS) that has evolved over a number of years and in a largely undirected manner. It was started with an inspired set of architectural and design choices, but its subsequent evolution has not been centrally controlled or directed. However, the many SoS-like systems that have been and are being proposed are all needed for specific purposes, and must be obtained on defined schedules and for committed costs. Examples include the U.S. Department of Defense FORCENet, Joint Battlespace Infosphere (JBI), and Future Combat System (FCS).

The technical challenges of developing, evolving, or otherwise composing SoS-like systems are impressive, as are the anticipated systems capabilities [Lewis 04]. Also as noted in Section 1 of this report, the characteristics of such systems differ widely from those of prior large-scale systems, and the development methods used in the past are almost certainly inadequate for this challenge [Fisher 06, Northrop 06]. The principal characteristics of these future systems-of-systems relate to their emergent behavior. Such behaviors stem from the systems' many largely autonomous nodes that must cooperate to produce complex and unique overall system capabilities.

In Section 2 of this report, the nature of future systems design is discussed, particularly regarding the partitioning of truly massive systems-of-systems into multiple semi-autonomous evolving large-scale systems. The tradeoffs required for this partitioning are particularly critical and have significant implications for the systems-development processes of the future.

Section 3, on critical system-development problems, makes four points. First, with few exceptions, the reasons that large-scale development efforts have failed in the past have not been technical. These projects have almost always failed because of project-management problems. Second, the solutions to these project-management problems are known and have proven to be highly effective, but they are not widely practiced. Third, if these known and understood project-management practices are not promptly and effectively adopted, future large-scale systems development programs will be unmanageable. Then, such systems will no longer be delivered late, over cost, and with poor quality; they will likely not be delivered at all! Finally, Section 3 also addresses the relationship between processes and technology, why both are essential for project success, and how the use of sound processes can actually improve technical and creative work.

Section 4 of the report discusses the requirements for a process that is scalable from small projects to truly massive development programs. It also describes why the current commonly used systems-development methods are not scalable and what is required for a process to scale up to support very large-scale work. This section also discusses the various ways large-

scale SoS-like systems are obtained, including through new development, the reuse of existing system elements, or even through an almost haphazard and uncontrolled assembly of independent systems and system elements into a cooperative SoS community. While SoS-like systems will be created in a wide variety of ways, it is important to consider the process and project-management implications of these various construction paradigms. Section 4 concludes with a brief discussion of the likely implications of these newer SoS-composition paradigms for process and project management.

In Section 5, the report describes the quality principles upon which a scalable process must rest, and discusses why these principles are both necessary and effective. It also discusses some of the quality properties to consider when composing large-scale SoS-like systems of existing products that may not have acceptable or even known quality characteristics.

Section 6, on future process needs, briefly reviews the nature of the project-management problems currently faced by essentially all large-systems development efforts, explains why many commonly used approaches are ineffective, and examines why attempts to scale up current methods to truly large-scale systems are destined to fail. Improved tools and technology are essential, but these advances must be coupled with effective processes and practices, or they can actually damage—rather than improve—development performance. Section 6 concludes by describing newer development methods that have been shown to work, and discusses why these methods are likely to be capable of scaling up to support very large-scale work.

Section 7 describes the strategic issues of scaling up newer process- and project-management methods to large-scale development programs. It discusses various aspects of these issues and describes approaches for testing the application of these methods to the support of SoS development and evolution programs. Section 7 closes with a brief review of the open issues and questions that must be addressed before truly robust process and project-management practices can be defined for working with large-scale SoS-like systems. Section 8 completes the report with a review of the process considerations involved in supporting a large-scale development program.

In conclusion, this report points out that continuing to develop the large-scale systems of the future with the methods of the past will almost certainly produce results that are much like—or even worse than—the typical results of the past. It also explains why attempting to solve these problems solely with improved tools and technology will be fruitless and possibly even dangerous. While the current focus on technical solutions is clearly justified by the enormous technical challenges ahead, an exclusive focus on technical issues without comparable efforts on program, process, and quality management issues would be a serious and expensive mistake. Unless the steps outlined in this report are taken in conjunction with continued technical research and development, the systems development efforts of the future will continue to fail, and often catastrophically.

Abstract

Some systems have some but not all properties of a system of systems (SoS). We refer to these as *SoS-like systems.* This report reviews the fundamental process and project-management problems of large-scale SoS-like programs and outlines steps to address these problems. The report has eight sections. Section 1 summarizes current thinking on the nature of future complex systems, and Section 2 discusses the systems-design problems of the future, particularly the partitioning of massive systems into system-of-systems structures. Section 3 points out how large-scale systems development efforts have typically failed because of project-management and not technical problems, and that the solutions to these problems are known and highly effective, but not widely practiced. It explains why, if the project-management problems of the past are not promptly and effectively addressed, large-scale systems development programs will likely be unmanageable. Section 4 discusses the requirements for a scalable process, and Section 5 both reviews and explains the quality-management principles upon which any scalable process must rest. Section 6 reviews the nature of the project-management problems currently faced by large-scale software-intensive system development efforts and explains why attempts to scale up current methods to very large-scale systems work will almost certainly fail. Section 7 describes process strategies for supporting development of a network-like system of systems and it outlines the process and project-management topics needing further research and development. Finally, Section 8 reviews the process considerations for supporting the very large-scale integrated development programs of the future. The report concludes that, unless steps like those outlined in this report are taken in conjunction with continuing technical research and development, the large-scale systems development efforts of the future will almost certainly fail, and often catastrophically.

1 Future Large-Scale Systems

Large and complex computer-based systems are now critical to the economic and military welfare of the United States and to much of the industrialized world. These systems form the backbone of modern military, business, economic, and governmental operations, and, without their continued support, our societies would be severely inconvenienced and even threatened. While the Internet is probably the best-known example of such a large-scale system, similar systems are now envisioned for addressing many future government and industrial needs. These so-called systems-of-systems (SoS) are characterized by emergent properties. These are properties of the entire system that are not contained within or supplied by any single—or even small group of—system nodes. Examples of such properties are performance, security, safety, and reliability. A system's emergent properties can also produce negative consequences, such as cascading failures, deadlocks, communications problems, denial-of-service attacks, weapons targeting degradation, or loss of command and control situational awareness.

An SoS is characterized by large numbers of autonomous interoperable nodes [Fisher 06]. Because these systems are often massive and because their properties cannot be completely and precisely anticipated, they are generally evolved over time, rather than being designed and developed as single centrally controlled monoliths. Further, because these systems are often widely distributed and because they typically must provide continuous essential services, they cannot be shut down. Therefore, these systems must be operated, maintained, repaired, and even developed while in continuous operation. As a consequence, all of the people, processes, and facilities involved in the operation, maintenance, repair, and development of these systems also must be viewed as parts of the system and considered in its development and operational processes.

The properties of these large-scale SoS are particularly significant for development. For example, there generally is no overall central authority, the system requirements constantly change, and the requirements may never even have been fully defined. This means that traditional requirements-based development practices are not suitable for such systems. In cases of long-lived platforms like weapons systems, no new node or platform can be permitted to harm the performance of the existing platforms. Since these systems are composed of largely autonomous nodes, these nodes or platforms become the focal point of systems management and must have properties that enable effective system operation, even when some nodes are inoperable, others are changing, and even if some are malfunctioning. Among the most important node properties are the following [Fisher 06].

- The nodes are loosely coupled. That is, there are relatively few dependencies among the nodes, and system operation depends largely on asynchronous communication among nodes.
- The information communicated among the nodes must be trustworthy. For effective system behavior, the nodes must provide timely, accurate, and properly defined information when and where needed.
- The nodes must be capable of operating in the face of threats and failures. That is, when one or more nodes fail or there is a system intrusion, the nodes must be capable of self-protection and recovery.

While this last property may appear to contradict the trustworthy requirement, the issue is system scope. That is, there must be some defined scope to the system and some way to determine when a communication is from a system member or an outsider. This is one of the unsolved problems with such systems: How does the system function when member nodes do not behave in accordance with the system's rules and constraints? There are many other such unsolved problems in SoS design and development. For example, the Internet is vulnerable to intrusion by untrustworthy parties, as demonstrated by the large volume of spam, worms, and viruses that currently plague its users.

2 The Design of Large-Scale Systems

The design of the large-scale systems of the future must involve a number of complex and challenging tradeoffs. The reason is that these systems-of-systems have major advantages as well as critical shortcomings. Among the advantages are the following:

- robustness in the face of node failure, maintenance, enhancement, or total replacement
- evolutionary development with multiple stable system-performance levels
- cooperative behavior of multiple independent or semi-independent system elements with limited central control and minimal dependence on the costs and schedules of node development and enhancement
- incorporation of systems functions that had not originally been visualized with new systems nodes or platforms

Conversely, the potential disadvantages of such a system-of-systems structure directly mirror its advantages.

- higher development and operational costs due to the redundancy required to sustain the autonomous nature of the system nodes and the impossibility of optimizing system-wide development or operational costs
- reduced system performance due to the necessity of communicating through the loose coupling imposed by the communications-centered system architecture and the impossibility of optimizing system-wide performance
- the reduced ability to secure the system against intrusion coupled with the absence of central system control
- designers' lack of requisite knowledge regarding how their new systems nodes or platforms will affect the entire SoS

Since most modern systems, particularly those needed for the military, must be both highly robust and efficient and secure, it is clear that the design of such systems must involve a host of complex tradeoffs between the robust and evolutionary nature of an SoS architecture and the cost, security, and efficiency of a centrally controlled monolithic system structure. This topic is discussed in more detail in Section 7.

3 Critical Systems-Development Problems

The critical systems-development problems of the future can be grouped into four classes.

1. cost and schedule
2. requirements instability
3. system properties
4. process management

The following paragraphs discuss these problems and their implications for SoS-like systems. An *SoS-like system* is one that has many but not all of the properties of an SoS. Additional comments on these issues and potential ways to address them are included in the final parts of this report.

3.1 Cost and Schedule Problems

Historically, the most intractable problems in developing large-scale systems of almost all types, and particularly for software-intensive systems, have been the inability to predict or manage development costs and schedules. Such systems often take years longer than scheduled to develop and cost far more than committed. While these problems have been severe in the past, they have not been fatal. That is, although these systems have taken longer and cost much more than planned, most of them have eventually been completed and fielded.

In the future, the cost and schedule problems for large-scale systems development are likely to be much more severe. The reason is that the development cost and schedule are important tradeoff parameters in system design. Because of their properties, SoS-like systems must have highly redundant designs. This means that their development cost and schedule estimates will likely be substantially greater than for comparable traditionally structured systems. That is why sponsoring governmental or industrial organizations will likely attempt to develop all or major portions of such systems as monoliths when they should be developed as collections of smaller semi-autonomous nodes in an overall SoS. Then, when the actual development costs and schedules for these massive systems are substantially greater than planned, the systems will likely be cancelled. The reason is that, in the past, cost and schedule differences have only been for a few years or millions of dollars. With these future systems, however, schedule variations could be of the order of decades and cost variances might run into billions of dollars.

With the pace of modern technology, any such large monolithic system that was a decade or more late would be obsolete before it could be fielded, if it ever could be. While customers

have been remarkably tolerant of poor performance by the systems-development community in the past, this tolerance is wearing thin, and such systems are increasingly being cancelled well before delivery. For example, the recent FAA en-route air traffic control system was cancelled after an expenditure of several billion dollars and a recent FBI intelligence system was cancelled after several hundred million dollars and several years of delay [Gross 05, GAO 99].

For SoS-like systems, the nature of the development cost and schedule problems will depend on the kind of system being developed. For systems such as the Internet, the system's ultimately most important properties will probably not have been planned or even anticipated at the outset. In these cases, cost and schedule may be important at the very beginning, but they would probably not be much of a concern thereafter. However, when such systems are intended for a specific purpose like battlefield management, then most of the system's capabilities must be available before the system can perform its targeted mission. In these cases, the fact that such systems are composed of multiple independent nodes will necessarily result in their being developed by multiple essentially independent projects. Further, since such systems will rarely be built entirely from scratch, almost all such SoS-type development efforts will include substantial prior-system content, either through the extensive use of commercial off-the-shelf (COTS) components or through enhancement of existing operational systems. This means that producing such a system, rather than involving one massive development effort, would typically involve many smaller largely independent development projects. This is a potentially severe problem for SoS-like weapons systems where changes in one node or platform could have cost implications for other platforms. Unless the overall system interfaces are comprehensive, precisely defined, and rigorously managed, serious unanticipated cost issues are likely to first show up in systems testing or even field operations.

A program with multiple independent projects would generally have less severe development cost and schedule problems than a single monolithic project with comparable functions. However, since such systems would generally be massive, the individual node-development efforts would also be large, at least by historical standards. Therefore, while the cost and schedule problems for an SoS-like system will likely be less than for an equivalent monolithic system, they would probably still be at least as severe as they are today.

One mitigating factor for problems with individual nodes of SoS-like systems is that the independent nature of these nodes would permit degraded operation of the system, even when most of the system's parts were not operational. However, if the objective was to produce a coordinated battlefield management system, for example, the fact that the individual weapons could still operate independently would not provide much benefit to the fighting forces. After all, they had that capability before the SoS program even started. So, since the eventual system's customers could still be unhappy, cost and schedule management will still be a significant problem for at least some SoS-like systems, and these problems will likely be severe if there are significant inter-node dependencies.

3.2 Requirements Instability Problems

While cost and schedule historically have been the most serious systems-development problems, requirements problems will be increasingly important in the future. The current common development paradigm attempts to freeze the requirements before building the system. This has never been an appropriate strategy, but it will be completely impractical in the future. A development process is required that can dynamically respond to changing requirements. In the past, this has principally been a planning and management control issue but, in the future, it will also be a technical issue. The reason is that the degree of requirements instability defines the required development schedule and resources, and the schedule defines the maximum feasible system scope. Here, system scope refers to the scope of the system nodes within the overall SoS. The logic for this conclusion is the following.

1. To have any hope of developing a usable system, all of the requirements uncertainties must be properly resolved.
2. The most effective way to resolve many of these uncertainties is to build a system version or prototype to test with representative users.
3. The larger the system scope, the longer these development and test cycles will take.
4. If the requirements change before the prior development-and-test cycle is completed, the entire development process can easily become unstable.

The responsiveness and predictability of the development process, coupled with the expected rate of requirements change, will define the upper limit for the size and scope of the system nodes in an SoS structure. This means that the more unstable the system-node requirements, the more important it is to limit node size and to minimize coupling and dependencies among the nodes by establishing system-wide interfaces.

3.3 System Properties Problems

The business of complex systems development has evolved through two phases and is just now entering a third phase. Phase One, the feasibility phase, started shortly after World War II and ended in the mid 1960s with the fielding of IBM's System 360 and the OS/360 operating system. During this initial rudimentary phase, the critical questions concerned feasibility—could we get these systems to work?

The second phase is the manageability phase, which is now gradually tapering to a conclusion. Here, the critical questions concern our ability to predictably develop and field such systems for anything near their committed costs. While this phase is far from completed, we are just beginning to embark on the more mature and challenging third phase, the quality phase. Here, the issues will concern the degree to which massive systems can reliably, predictably, safely, and conveniently do what their users need them to do. Because the systems now being envisioned will have a fundamental and pervasive impact on the way much of the world's population lives, works, plays, and fights, system properties will be critically important.

While extendibility, maintainability, privacy, reliability, safety, security, usability, and many other properties have long been recognized as important, they have not yet become sufficiently important to drive the marketing and business processes that govern development priorities. As long as development cost and schedule issues remain largely uncontrolled, the system-property issues will continue to be viewed as secondary. But once the current cost and schedule problems appear to be under control, these now-secondary issues will become primary. At that point, a governing parameter in the development process will become quality. The reason is that a poor-quality system cannot predictably be extendable, maintainable, private, reliable, safe, secure, or usable.

3.4 Process Management

Because of software-development's early history as a largely individual creative activity, there is a general attitude among software academics and professionals that disciplined processes limit creativity and are impractical and unnecessary for developing software-intensive systems. Further, because even very large software products have occasionally been produced without the kind of disciplined processes and extensive design specifications required for other large-scale activities, software work has never developed the mature process- and project-management practices needed to predictably develop high-quality large-scale products.

While experiences with the Capability Maturity Model® Integration (CMMI®) and Team Software Process SM (TSP SM) have shown that such outdated attitudes can be overcome when an organization's management is convinced of the value of disciplined processes, an orderly and planned improvement strategy is essential for the truly large-scale work required for SoS-like systems. In these cases, there will not usually be the single monolithic management structure needed to make process improvement a priority. Unless the prevailing attitudes of the software community can be changed, future SoS-like development efforts must generally rely on the process- and project-management methods of the past. This is unfortunate because many of these systems will then almost certainly be expensive failures.

There are five ways in which a defined, planned, measured, and quality-controlled process can help improve both the business and technical performance of very large-scale traditional and SoS-like programs.

1. All modern science and engineering is based on learning from prior experience. Competent engineers and scientists know what experiments have been successful and base their processes on them. They stay current with process research developments and do not waste time experimenting with processes that have already produced unsatisfactory results. Until developers consistently use defined and proven processes, they will waste their time relearning known truths.

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
SM Team Software Process and TSP are service marks of Carnegie Mellon University.

2. With an inefficient or ill-defined process, developers must follow poorly defined and inaccurate plans. With the data available from a modern process, plans can be both accurate and precise. With more appropriate plans, developers do not waste their time on routine project-management and planning tasks and can devote more of their efforts to creative technical work.
3. Quality work is not done by mistake. Quality must be planned, measured, tracked, and managed. When it is, product defect levels are normally reduced by orders of magnitude [Davis 03]. With current common practices, large software products typically have thousands of test defects and developers spend at least half of their time finding and fixing enough defects for the product to run the basic tests. Even then, finished products have many unidentified defects. While it takes considerable skill to fix defects in test, fixing defects is not creative work. For SoS-like systems, the development work must be done by people who strive to produce quality products in every step of their work.
4. Development is a learning process. However, unless this learning is codified and preserved, the resulting knowledge is generally lost. That is the reason developers should define, use, and continually improve their processes: to build on their own and other's experiences.
5. For individuals and groups to work together effectively, they must coordinate their activities. While very small groups can often do this informally, without defined processes and detailed plans, large groups cannot.

To have a reasonable chance of being successful, the more challenging SoS-like development programs of the future must address the critical systems-development problems of cost and schedule, requirements instability, system properties, and process management. Further, whenever the separate system node or platform development efforts must coordinate their work, a higher level of system coordination, process management, and development planning is essential. Later sections of this paper describe the actions required to provide such capabilities.

4 The Scale-Up Problem

Developing truly large-scale systems necessarily involves many critical problems. For example, such systems programs should follow an adaptable design strategy, use incremental development, and adopt an architecture that permits the various sub-systems to evolve independently. These, however, are largely technical issues, and current research is considering them [Fisher 06, Northrop 06]. The fundamental process-management problem in developing the large-scale systems of the future is that the current commonly used development methods are nearing their scalability limits and are incapable of handling the much larger challenges of the future. This is because the process scale-up problems have yet to be addressed in an orderly way. We've seen that the technical scalability issues concern the analysis, specification, design, and implementation methods for developing progressively larger sized systems. The process scalability issues concern organizing, planning, measuring, tracking, reporting on, and controlling the work. Process scalability has three principal elements:

1. method scalability
2. management scalability
3. measurement scalability

4.1 Method Scalability

The method scalability problem concerns both the need for new methods as system scale increases and the suitability of the methods used to build the supporting smaller scale foundation on which the larger-scale products depend. One such method, for example, would be quality management. For very small programs, merely compiling and testing usually produces running programs in a reasonable amount of time. As system size increases, groups generally find that personal reviews, team inspections, and independent testing are needed. With further scale increases, statistically based quality-control methods are required to consistently produce quality products.

The need for new design, analysis, project-management, and quality methods with increasing scale are obvious in building construction. For example, in building a two-story house, one must worry about the sheer-strength of the first-story walls. This is not a serious problem with a one-story house but, with a second story, the wind has the leverage to blow the first story over sideways. Further, when building height is extended to dozens of stories, these sheer-strength problems are greatly magnified. This example also illustrates the second method scale-up concern: the continued suitability of the smaller scale methods as the pro-

ject's scale increases. SoS-like systems are like skyscrapers and the suitability of all of the methods used in their design and development must be defined, managed, and quality controlled. The house-building example also suggests a third scale-up concern: when a project involves a system of unprecedented scale, that project must discover for itself the need for new methods. Finally, the fourth and most important scale-up issue is that, when projects of unprecedented scale do not use the best known methods, they will generally be so consumed by traditional problems that they will not even recognize the need for new methods.

With software-intensive systems, considerable attention is often paid to the newer methods needed by increased system scale, but there is generally little or no focus on the continued suitability of the existing smaller scale methods. The principal concern here is with the design and quality-management methods used in software development. When developers design and implement the modules and components of massive systems, they typically continue to use precisely the same methods they learned to use with the small toy problems they solved when first learning to program.

While experienced developers will generally include provisions for recovery and error-handling, and they may even build in test provisions, their basic design and quality management practices are essentially unchanged whether they work on a 100 LOC program fragment or develop a part for a 10,000,000 LOC system. The lack of an analyzable design and the continued reliance on testing as the principal quality-management method are fundamental method-scaling problems. These problems must be addressed if there is to be any hope of scaling up development processes to handle the truly massive systems of the future.

4.2 Management Scalability

The management scalability problem concerns the estimating, planning, assignment, tracking, reporting, control, and system-wide coordination of development work. The problem here is that, with current common practices, the managers make the plans, allocate the work, report progress, and exercise project control. This might not be a serious problem if these managers understood the work, knew how to make accurate plans, and could make optimum work assignments. Unfortunately, however, in most organizations the level of available pertinent project information is inversely proportional to management level. Therefore, when organizations follow the principles of hierarchal authority, the views of the higher level managers are given greater weight in making the project planning, assignment, tracking, reporting, and control decisions. This means that the suitability of many of these decisions will be inversely proportional to the level at which they were made.

While the extent of this hierarchical-management problem varies among organizations, it is common. Therefore, when organizations use management-centered planning, allocation, tracking, and control processes, their management systems are inherently incapable of scaling up to support truly massive systems-development programs.

With SoS-like systems, the management scalability problem can be substantially reduced by the inherent delegation resulting from the system's architectural structure, but only when the separate node or platform development efforts are truly independent of one another. However, as the sizes of the system's nodes grow or as the level of interaction among the nodes increases, the management scalability problem will again arise. Therefore, while the management scalability problem will be reduced with SoS-like systems, it will not be eliminated and should be considered when deciding how to make system-wide decisions.

4.3 Measurement Scalability

Why is it that financial accounting methods work well for very small organizations and are equally effective for very large corporations? The fundamental reason is that the financial community uses precisely defined methods and auditable data. These methods also presume that these data can be in error, so they include consistent auditing and checking methods to ensure that the data at every organizational level are both accurate and precise. Few of the methods commonly practiced in the development of software-intensive systems use data of any kind, and those that do so typically rely on data gathered by staff, support, and service groups such as accounting, testing, and field support.

To make accurate plans and to precisely determine project status, the methods used throughout the organization must rely on consistent and statistically useful measurements instead of after-the-fact guesses and approximations. When development processes use such robust methods, they at least have a chance of being scalable. However, until they do so, the development methods and management practices will continue to be inexact and approximate, and the cost, duration, and quality of system-development work will be progressively less predictable and more troublesome as programs become larger. This will be particularly true for those SoS-like systems that grow and evolve over time. The reason is that their overall characteristics and behavior will not generally be understood or even known in advance so a top-down commitment-based management system will not be appropriate.

In summary, for any large program or organization, the methods used at the top, at the lowest working levels, and at all levels in between must be precise and robust. If they are not, the process will not be scalable. In making technical and project decisions, the management system must be based on and give great weight to the knowledge and judgment of the development-level professionals. Furthermore, the decision process must consider all pertinent sources of information, including the developers, managers, architects, users, and anyone else who could usefully contribute. Finally, all of the program's management, technical, and quality practices must use data that are derived from accurate, precise, and auditable process and product measurements

5 Quality Principles

The governing quality consideration for large-scale systems development is that a high-quality process will consistently produce high-quality products, while a poor-quality process will generally produce low-quality products. The problem here is with the word “generally.” People tend to remember their occasional successes and forget their less successful achievements. As a result, when a development group has produced a seemingly high-quality product with an unmeasured and poorly controlled process, the members tend to feel that they have proven that process and should continue to use it for future work. However, unless they have measured and statistically verified that this process consistently produces quality products, they run a significant risk of getting poor-quality results.

With really massive monolithic systems, even a small chance of getting any one poor-quality component would compound to almost-certain quality problems for the overall system. Such systems are usually built as single monoliths instead of as collections of independent nodes in an SoS because of the potential for increased performance, security, economy, or some other property that could not be obtained by decoupling the system into relatively independent elements.

However, with SoS-like systems, there are two categories of quality to consider: node quality and architectural quality. First, the quality issues for the system’s multiple nodes should be substantially less severe than those for comparable traditional systems. This is because the quality issues for the system nodes are generally more manageable due to their reduced size and relative independence. The architectural quality category concerns overall system behavior. SoS-like systems must have some overall architectural framework that enables the system nodes to exchange information and to cooperate. The sophistication and complexity of this overall architecture will then depend on the sophistication required of the interactions among the nodes. This in turn will determine how tightly the nodes must be coupled.

For example, a highly secure SoS-like system must have comprehensive and robust provisions for verifying the sources and content of all intra-system communications. This architecture must then be embodied in the overall system-wide design and implemented in every system node. Any quality problems in the architecture, design, or implementation of these security provisions could then impact the security of the overall system. Similar concerns will apply to safety, privacy, performance, usability, or any other system-wide characteristic. Furthermore, when some system property unpredictably emerges during system operation, new system-wide quality concerns could appear. This, for example, is the case with Internet security. Security was not an initial Internet design concern and buffer-overflow defects were not a serious issue. However, with the widespread use of the Internet, security has become a ma-

jor issue and buffer overflows are responsible for about half of the known Internet security vulnerabilities [Kay 03, Seacord 06].

The tight-coupling characteristics of a system generally result from optimizing its overall design and from minimizing redundancies and inefficiencies among its component parts. This in turn results in far closer coupling among the system's components and large numbers of critical interdependencies. The quality methods required for a process to be scalable up to truly large-scale work are based on three critical assertions which are summarized and discussed in the following paragraphs.

1. To consistently produce quality products, the process must have the objective of removing all defects before test entry.
2. To ensure an effective and high-quality process, that process must provide quality measures.
3. To effectively manage a quality process, everyone from the senior executives to the individual systems-engineering and software-development team members must support and participate in that quality process.

For SoS-like systems, these assertions apply equally to node and architectural quality.

5.1 Removing All Defects Before Test Entry

While removing defects before test entry is generally viewed by the software-development community as a radical objective, this objective is the foundation for all quality programs in all other fields of science and engineering. The logic for this position rests on three proven facts.

First, no test yet devised can identify more than a fraction of the defects in a large and complex product, and the larger and more complex the product, the smaller this fraction becomes. This means that to get a high-quality product out of testing, one must put a high-quality product into testing. This further means that any process that consistently produces high-quality products must remove essentially all of that product's defects before the start of testing [Humphrey 02].

The second proven fact is that the later in the process defects are detected and removed, the more it costs. For example, with systems-level testing, it typically takes an average of several hours to find and fix each product defect. Conversely, statistically managed team-level reviews can typically find and fix five to ten defects per hour, and team inspections can find and fix one to two defects per hour [Humphrey 05].

The third proven fact is that even experienced and competent software developers inject many defects into their products. It is not that they are careless or incompetent, just human. For example, the Software Engineering Institute (SEI) has gathered data on several thousand software developers, and these data show that experienced development professionals inject an average of over 100 defects per 1,000 lines of developed code [Humphrey 05].

While this number may sound high, it is important to realize that 1,000 lines of code represents about 30 pages of printed text or program listings. This means that average defect-injection rates are a little over three defects per page, including logical and typographical errors, and almost all of these defects must be found and fixed before the program will work. This means that the quality of what is commonly viewed as defective computer software must be much better than that of other human-created products.

Experience shows that when the developers rely entirely on compilation and testing for defect removal, their products have approximately 5 to 10 remaining defects per thousand lines of code at system test entry. This is about one defect in every 3 to 5 listing pages. Therefore, even a modest-sized product of 1,000,000 LOC would typically contain 5,000 to 10,000 defects at systems test entry. Finding and fixing these defects at a cost of several hours each can be expensive and take a very long time. For example, with the first release of the 6 million line-of-code NT system, 250 Microsoft developers spent a year finding and fixing 30,000 defects [Zachary 94]. That is an average of 16 hours per defect!

With current common development methods, even modest-sized software products contain many defects, and any process that relies exclusively on testing to find and fix them will spend a great deal of time in testing and still deliver defective products. Such a process is too costly, too time-consuming, and too inefficient to be relied upon for developing the large and complex systems of the future.

5.2 Quality Measures are Essential

Today's commonly used software-development processes do not incorporate quality measurement and analysis. This is a crucial failing since, with even rudimentary measures, the above-named facts would be obvious and sounder and more efficient software-quality practices would have been adopted long ago. The simple fact is that without measurements, no serious quality program can be effective. While developers can make rudimentary quality improvements without measures, achieving the defect levels required in modern complex systems must involve defect levels of a very few parts per million. Such levels are not achievable without complete, consistent, precise, and statistically based quality measurement and analysis.

5.3 Everyone Must Participate in the Quality Program

The requirement that everyone must participate in the quality program is a direct consequence of the previously stated facts. To achieve defect levels of a few parts per million, quality must be a high priority for everyone. Quality work is not produced by mistake; it results only from a consistent striving for perfection. The individual development team members must all strive to produce defect-free work. Any undisciplined work by any systems engineer, software developer, tester, or almost anyone working on or with the product can be a

source of defects, and their work must be measured and quality controlled. If it is not, high-quality products simply will not be produced.

This is not only true for all of the development-team members; it is also true for all of the support groups, managers, and executives. For example, an executive decision to skip some review, test, or check because "We don't have the time" will inevitably lead to sloppy work and poor-quality products. The only acceptable attitude at *all* levels of the organization must be "We don't have time to do it wrong!"

5.4 Quality Principles for SoS-Like Systems

While the above quality principles apply to single systems, there are some special considerations for SoS-type programs. Here, for example, the relative independence of the nodes means that overall-system performance will not be impacted by many of the individual-node quality problems. The exceptions will be where some emergent function depends on the behavior of one or more poor-quality nodes. This means that the quality principles for SoS-like systems reduce to the two categories of quality previously discussed: node quality and architectural quality. In most cases, SoS quality could be considered at the node level. However, where system performance depends on the proper performance of multiple nodes, quality management must involve the quality properties of all of the involved nodes as well as the relevant aspects of architectural quality. In these cases, the issues of poor node quality could become very important, even for SoS-like systems.

One case where quality issues for SoS-like systems could be more challenging than for monolithic systems is where some one or more defective nodes produces spurious results that adversely impact the overall system. Because of the likely lack of central system control, such spurious effects could be hard to pinpoint and fix. In one personal example, AOL blamed my email problem on Dell, Dell blamed it on Microsoft, and Microsoft blamed it on AOL. I solved the problem by moving to gmail. Presuming that such problems could be identified, the repair of these defects would require work by those responsible for the defective nodes. Then, unless some way could be found to convince these distributed groups to do the required work, SoS performance could be adversely affected.

So, while quality management may not initially appear to be a serious SoS issue, nothing is free. It is, for example, likely that future SoS-like systems will have emergent quality characteristics that are heavily influenced by the quality of the individual nodes. Here again, the Internet affords an excellent example. To correct the Internet's security vulnerabilities and to counter denial-of-service attacks, every Internet service provider will ultimately have to make major changes. Furthermore, all of the Internet-connected node-development work will have to view security defects as having high priority. Another example would be the inclusion of a COTS product where, because of the new and unanticipated environment, latent COTS quality issues were exposed. Unfortunately, due to the nature of these SoS-like systems, such problems would not likely appear until the systems were operational.

The Internet's security history provides a useful quality lesson for future SoS-like systems. Over 90% of all Internet security vulnerabilities are caused by standard programming defects and over 50% of these defects concern buffer overflows [Seacord 06]. While buffer overflows have always been recognized as defects, they were not important as long as no one consciously tried to cause them. However, once the Internet's vulnerability to buffer overflows was recognized, a new threat emerged, and this threat is almost entirely due to the poor quality of the software in the systems using the Internet. While the original Internet architecture could been designed to anticipate and counter this quality problem, it was not originally considered important. It seems likely that future SoS-like programs will encounter similar latent quality, security, safety, performance, usability, or other system-wide problems.

5.5 Quality Considerations for a Systems Composition Process

When a new system is composed of existing COTS products or built from small modifications to existing products, there are some special quality considerations. First, the quality characteristics of these system parts may be completely unknown and there may be little or no way to get the cooperation of the product owners in fixing quality problems. Second, these component parts may be proprietary and their internal structure not available, making it difficult if not impossible for anyone else to the fix problems, even if they could be identified. Third, in large SoS-like systems that are composed in this way, it may even be difficult to identify the sources of quality problems.

While this may seem to be an impossible set of conditions, the independent nature of the nodes in SoS-like systems provides an opportunity that is not otherwise available. COTS product suppliers presumably have competitors, so there may be alternatives available if one product were found to be seriously defective. Since such a quality problem would probably be discovered after system deployment, the component-replacement problem should be considered during system construction. Then, if replacement were practical, defective COTS parts could be replaced or the threat of replacement could be used with component suppliers to get defective parts repaired.

In SoS-related work it is essential to employ sound quality-management principles during system operation and to keep extensive data on component performance. Then it should be possible to use these data to decide to replace or repair some system element. Where the internal structure of the COTS products is known, these data can also be used to identify and repair or replace the most defective modules of these products.

These actions require that there be some central resource for monitoring system performance, gathering quality data, and initiating repair or replacement actions. Where such a resource is not available, the quality problem reduces to a node-centered quality problem that must be handled with traditional single-system quality-management methods.

6 Future Process Needs

As noted in the previous sections of this report, the development processes of the future must meet five requirements. They must

1. predictably control development costs and schedules
2. responsively handle changing needs
3. minimize development schedules
4. be scalable
5. predictably produce quality products

These topics are covered in the following sections.

6.1 Predictably Control Costs and Schedules

Cost and schedule problems are not new, and many other fields have learned how to manage them. The solution has always been the same.

- Have the people who will do the work estimate and plan that work.
- Precisely and regularly track the progress of the work.
- When progress falls behind plan, promptly identify and resolve the problem causes.
- When the requirements change, promptly re-estimate and revise the entire plan at all involved levels.
- Anticipate and manage risks.

While one might question the applicability of this approach to systems and software development, there is now ample evidence to demonstrate its efficacy. Nearly 20 years of experience with process improvement have shown that these principles, when applied at the node-development levels, can have important benefits [Chrissis 03, Humphrey 02, Paulk 95]. However, it is not yet clear how well these principles apply to SoS-like programs. Where an SoS-like program was composed of multiple independent nodes, it is clear that these principles should apply to the development and management of the individual nodes. However, it is not as obvious that these methods should be applied to the overall system-wide architectural and engineering effort.

In addressing this question, there are two principal considerations. First, any central architectural and engineering effort would almost certainly have to operate within some set of cost, schedule, and requirements constraints. Since any such effort must presumably be managed,

traditional cost and schedule management principles should apply. Where this central architectural and engineering effort were small, the management problems would then presumably be much like those of similar-sized traditional development programs.

The second consideration relates to the broader systems context for the central architectural and engineering effort. Here, as long as the central work did not impact or depend on the work done at the nodes, the central effort could be managed like a small stand-alone project. However, where there were interdependencies, the central effort must consider the impact of its actions on all of the nodes. Similarly, each of the nodes must consider the system-wide impact of its own work. Here, recognized and proven requirements-management and quality-management principles should almost certainly be applied across the entire SoS complex. While the areas requiring such broad-gauge management might not be numerous, where they arose, the impact could be severe. In these cases, the following logic should apply.

- Whenever the system is sensitive to a node-development effort, that effect must be identified and steps taken to provide SoS-wide technical and management coordination and control.
- Similarly, any sensitive project in a sensitive node must also be identified and steps taken to provide SoS-wide technical and project management coordination and control.
- Also, any sensitive team in a sensitive project development effort must also be identified and steps taken to provide SoS-wide technical and team management coordination and control.
- Since the actions of any individual developer on a sensitive development team could cause systemic problems, steps should be taken to identify these team members and to include their work in the SoS-wide technical and team-management activities.
- Finally, for individual systems engineers and software developers to participate effectively in project management, they must plan and manage their personal work.

Three objections are typically raised to this logic. The first is that systems and software development is creative knowledge work and that precise planning and project management would have an adverse effect on creative people. This has not been a problem in other fields of engineering, science, or the arts, and there is ample evidence that it is not a problem for software development [Chrissis 03, Humphrey 02, 05, 06a, 06b]. To date, however, there is no similar evidence for systems engineering or for work on very large systems. Efforts are currently underway to demonstrate the effectiveness of these methods for systems-engineering work. However, there is currently no funded and staffed effort to apply these methods to large-scale development programs. It is hoped that this report will lead to such an effort.

The second objection is typically that software and systems-engineering people will be unwilling or unable to plan and manage their personal work. Again, the evidence to date shows that this is not the case for software developers. While some small percentage of software developers do object to planning, tracking, and managing their personal work, the number is

typically less than 10%, and even those people normally change their views after they have completed one or two projects with these methods [Davis 03, McAndrews 00]. Similar data are not yet available for systems-engineering work.

The third objection concerns training time. Training systems engineers and software developers to estimate, track, and improve their personal work takes a little time. However, most organizations soon realize that the required time investment is modest when compared with the schedule uncertainties of major programs.

To control the cost and schedule of even small development programs, it is essential that the development professionals plan and manage their personal work. The key to making these methods acceptable for the systems engineers and software developers is personal control [Humphrey 06b]. Once they get over their initial aversion to what appears to be administrative mechanics, most software developers find that self-management methods are surprisingly helpful, and that these methods actually enable them to be more creative rather than less so. While there are as yet no data on the reactions of systems engineers to these personal practices, it seems likely that the same conclusions will apply. Efforts are currently underway to demonstrate this for systems-engineering work.

6.2 Responsively Handling Changing Needs

While responsively handling changing needs would seem to be a simple issue, it is not. The problem is being responsive to new information while continuing to meet prior cost and schedule commitments. In fact, it is just this tradeoff that is responsible for many of the severe cost and schedule problems of large systems programs. To be responsive, development groups must do the following.

- Examine every proposed change to understand its effects on the development plan.
- Pay particular attention to each change's impact on completed work, including the requirements, design, implementation, verification, and testing activities.
- Estimate all cost and schedule consequences of making the needed change.
- Where the cost and schedule implications are significant or where they exceed the currently approved plan, get management approval before proceeding.

While these steps may seem simple and straightforward, the problem is that they must be strictly followed by every member of every development team throughout an entire program. Many requirements changes seem relatively minor to developers, and they will often agree to make them without much thought. However, without a careful analysis, some seemingly simple changes can have major consequences. Unless every developer maintains a personal plan and updates that plan for every change, this level of change management is impossible. However, once all developers maintain and update personal plans, it is relatively easy for teams and even entire development programs to be responsive to changing needs [Davis 03, McAndrews 00].

This way of handling requirements changes must be applied to all of the node-development programs as well as to any architectural or design changes that have systemic effects. Just as with cost and schedule management, all architectural and some node-level requirements changes must be examined and any that could have systemic effects must be managed according to these principles. While this should not be a major problem where the SoS structure is loosely coupled, it would likely be a significant problem for typical weapons system platforms which are generally tightly coupled.

6.3 Minimizing the Development Schedule

There are only three ways to minimize development schedules.

1. optimize the project staff
2. reduce the amount of work
3. minimize the amount of rework

6.3.1 Optimizing Project Staff

While project staffing is not a process-management issue, inadequate staffing can impact the ability of teams to properly follow their processes. Many development organizations attempt to run projects with either inadequate or part-time staffing. However, understaffing projects delays them and increases their costs. Understaffing also increases schedule pressure and often leads team members to do superficial or poor-quality work in an attempt to meet some critical date.

A common and very expensive alternative to understaffing projects is simultaneously assigning systems engineers and software developers to multiple projects. This is expensive because it takes them time to mentally switch between projects and to rebuild the prior context for their work. This switching process is also a frequent source of mistakes which increases both testing time and rework costs. Further, when professionals are not working full time on one project, they are often unavailable when other team members need them, so their teams cannot build the trusting and cohesive relationships that characterize high-performing teams.

Finally, systems engineers and software developers are creative people and, when they are immersed in an interesting project, they tend to think about it and to solve project problems when eating, driving, or even sleeping. This kind of creative energy is often lost when people have multiple part-time assignments.

Depending on the kind of work being done and on the availability of suitably skilled staff members, projects should be fully staffed whenever possible. While optimum staff size will vary with project size, type, and phase, maintaining an optimum project staff will ultimately reduce the development schedule and minimize project costs. This practice is applicable to all professional programs, both SoS and traditional.

6.3.2 Reducing the Amount of Work to be Done

Another obvious but highly effective way to reduce a project's schedule is to reduce the amount of development work to be done. While this does not necessarily reflect on the process being used, it can. For example, a decision to maximize COTS product use or to develop a new system by modifying and/or extending an existing system can have significant process implications. Since this will be a common strategy for most large-scale programs, the process must have provisions for exploiting both COTS product and system-enhancement opportunities and for performing the work so as to capitalize on these opportunities [Albert 02, Hansen 99, Sai 04].

With SoS-like programs, it may be possible to reduce the work required to field an initial system version by using some combination of COTS products and minimally modified existing systems. This strategy can also cut initial program costs, schedules, and risks. Ultimately, however, if major new system functions are required, the necessary development time and money must be spent. Further, since SoS-type programs must have highly redundant designs to permit independent node operation, the overall program schedules and costs will generally be greater than would be the case for a comparable monolithic structure. On the other hand, if the system is truly massive, it might not even be feasible to build a comparable monolithic system.

6.3.3 Minimizing Rework

Shortening the project schedule by minimizing rework is a process issue. While the process characteristics implied by this requirement are discussed more fully in the subsequent sections on quality, it is important to first discuss why reducing rework actually reduces the development costs and schedules for all types of engineering programs. For anyone who is familiar with traditional quality-management principles, the connection between high-quality work and minimum rework, and between minimum rework and reduced project costs and schedules, would seem too obvious to discuss. However, because these quality principles are not universally accepted, it is important at least to outline the logic that supports them. The following principles of quality management are based on facts that have been demonstrated in every field where they have been tested, including software [Deming 00, Humphrey 06a, Juran 88].

- It costs more to build and fix defective products than it would have cost to build them properly the first time.
- It costs more to fix a defective product after it has been delivered to users than it would have cost to fix it before delivery.
- It costs more to fix a product in the later testing stages than in the earlier design and development stages.
- It costs less to fix product requirement and specification errors in the earlier requirements and specification stages than in the later design, implementation, and operational stages.
- It is least expensive to prevent the defects altogether.

These quality principles are based on experience with large-scale monolithic systems, and they apply equally to SoS-type systems. The only point of debate concerns requirements defects. Here, however, the issue is not really one of correcting known defects as much as with understanding the system's true requirements and how to achieve them. Where the requirements are not clear, it is often best to make a first cut at the requirements and to then test the related function in practice. In such cases, an SoS-like system framework could provide a ready test bed for experimenting with potential new system features. Whenever the requirements for new nodes or platforms are known to be wrong, it is always cheapest to fix them at the earliest possible point in the process. However, when the requirements for a fielded platform are later found to be incorrect, the requirements and possibly even the entire system strategy may have to be reevaluated.

Even though these principles have been proven in every case where they have been tested, they are still not universally accepted. The reason is that organizations often establish separate groups for product development, production, testing, and field repair. Therefore, while the total organization would save time and money by following sound quality practices, the added costs of quality work are born by one group while the savings accrue to other groups. Also, quality programs typically increase costs in the early program stages and reduce them in later phases. These early investments in quality programs are generally hard to justify, particularly when organizations do not have the quality data that supports their introduction. Finally, unless managers and developers have personally experienced the benefits of an effective quality program, few are willing to make the necessary effort to do high-quality work. The distributed and largely autonomous nature of SoS-like-systems engineering programs will likely exacerbate these problems.

6.3.4 Being Scalable

The fourth requirement for a process to be suitable for large-scale system development is that it be scalable. As noted in Section 4, to be scalable, a process must meet the following three criteria.

1. It must use robust and precise methods at all levels, especially at the working systems-engineer and software-development levels.
2. For technical and management project decisions, the management system must be based on and give great weight to the knowledge and judgment of the development-level professionals and anyone else who has relevant information.
3. The process must consistently use data that are derived from accurate, precise, and auditable process and product measurements.

While there are potentially many ways to meet these needs, the best-known and, to the author's knowledge, the only currently available method that does so, is the Team Software Process (TSP) developed by the Software Engineering Institute [Humphrey 02, 06b]. The TSP practices used to meet these scalability criteria are shown in Table 1 and described in the paragraphs following.

Table 1: TSP Scalability

No.	Topic	Criteria	TSP Practices
1.	Design	Precise, complete, unambiguous	– Team-established design standards – Documented designs – Planned, measured, tracked design practices
2.	Teams	Teambuilding, support	– Trained and qualified team coach – Structured team launch – Team-defined process and strategy – Team-developed plans based on historical data – Negotiated team commitments
3.	Measures	Scalable measurement system	– Defined and precise process data – Team-member data gathering – Team data used in project planning and tracking – Data-based project management, quality control

The TSP does not define a specific design method, but it does call for development teams to establish and define the design standards they judge to be most appropriate for their projects. All team members must produce documented designs and use defined, planned, measured, and tracked development methods that ensure design and implementation quality.

The TSP management system provides team-building and coaching support for what are termed *self-directed teams*. These teams use data from their personal and team work to make their own plans and to track and control their own work. TSP teams also define the processes for their work, produce team and personal development and quality plans, and negotiate commitments with management.

TSP teams address measurement scalability by tracking the time every team member spends on every project task, measuring the size of every product produced, and recording data on every defect found in the requirements, specification, development, and testing processes. They then use these data in planning, tracking, and managing their work and in controlling the quality of the products they produce.

While consistently following all of the TSP practices takes training and personal discipline on the part of team members, many organizations are using the TSP and growing evidence supports its efficacy [Davis 03, Grojeans 05, McAndrews 00, Pracchia 04, Rickets 05, Trechter 05]. TSP suitability for very large-scale development work has not yet been demonstrated, but it is the only currently available and widely used method that meets the scalability criteria presented in Section 4. This author strongly suggests that a scaled-up version of the TSP be developed and tested with one or more large-scale systems programs.

6.3.5 Predictably Producing Quality Products

The fifth and final requirement for a process to be suitable for large-scale use is that it predictably produce quality products. The requirements for a process to do this are as follows.

- The process must include a family of early defect-prevention and defect-detection activities.
- To ensure that all or almost all product defects are found and fixed prior to the start of testing, the process must include measures that verify **product** quality, both before and after testing.
- The process must define measures that can be used to verify **process** quality during project planning, process enactment, and following process enactment; and these measures must be gathered and used by all the systems-engineering and software-development team members and their management.
- The process must ensure that quality plans are regularly produced and reviewed, and that deviations from these plans are promptly detected and corrected.

While these quality practices have been proven highly effective, there are typically three objections to adopting them.

1. In spite of ample evidence that demonstrates the efficacy of such quality methods, there is still a general skepticism regarding their effectiveness [Davis 03, Humphrey 02, McAndrews 00].
2. Groups that do not have data also commonly argue that early defect-removal methods only find the simple defects and that the more complex and hard-to-find defects are still left for testing. While this is a valid concern for defects in the emergent properties of SoS-like systems, these are likely to be relatively small in number. For the volumes of defects typically found in node development, no one has thus far provided any empirical evidence to support this contention.
3. Many software developers are convinced that they can find defects much more quickly with the aid of a debugger than they possibly could with personal reviews or team inspections. Again, the studies cited above did not find this to be the case, and no empirical evidence has yet been provided to support that position. Furthermore, systems engineers are typically willing to review their requirements, specification, and testing products, but few follow the data-driven review practices required for high-yield reviews [Maldonado 06].

In concluding this discussion of quality, it is important to stress the need to use a family of defect-prevention and defect-detection methods. Some defects are extremely difficult to find in any way other than with system testing. Conversely, system testing is a time-consuming and expensive way to find and fix the volumes of defects injected by most development projects [Humphrey 02]. Personal reviews and team inspections have been found to be much more effective and efficient [Humphrey 05].

Some defects are difficult to detect prior to system testing because they result from unexpected interactions among components. However, many of these problems can be prevented or removed early in the development process by using known design and quality-management methods. Furthermore, many of the defects not removed prior to system testing become as difficult, time intensive, and costly to find and fix as the truly problematic system defects. To effectively and efficiently resolve the system-level defects, all other defects must first be removed because defective components can mask system-level defects, making them extremely hard to find. Further, the defects found in system testing of SoS-scale systems are likely to be substantially more expensive to find and fix than those found in smaller systems. Therefore, where system testing is possible, effective testing of an SoS-like system will require even stronger components than are needed for traditional systems. When building an SoS, early defect prevention and removal practices are even more critical than with smaller scale systems.

For SoS-like systems, overall system testing is likely to be impossible and such systems are likely to grow through the largely uncontrolled addition of new and/or modified nodes. Where overall SoS performance is independent of node performance, this will not likely cause quality problems. However, where node properties can impact SoS-wide behavior, the quality of both new and modified nodes must be rigorously controlled or the entire SoS complex will be subject to frequent and unpredictable disruptions. This, for example, is exactly the case with security defects and the Internet.

7 SoS Process Strategy

If a group had to develop an SoS-like system, how would it best proceed, what practices do we now know that should be followed, and what would we have to learn before providing sound advice? This section addresses this set of questions. It is important, however, to recognize that these questions imply three contradictions with the points previously made in this paper. First, it has been asserted that SoS-like systems are not really developed; they happen. Second, it has been noted that SoS-like systems are not usually centrally controlled and managed. Third, many of the critical properties of SoS-like systems are emergent and cannot be anticipated or developed with a traditional top-down management process.

Given these factors, it would appear pointless to talk about either a development process or management methods for developing SoS-like systems. While these points are all valid, they miss one important issue: SoS-like systems have such attractive properties that many groups will wish to produce them, whether they know how to do so or not. So the real question is: If you must have a system that is to have SoS-like properties, how should you proceed? There are five parts to answering this question.

1. defining objectives and expectations
2. concentrating on the core system concepts
3. launching multiple parallel and largely independent efforts
4. experimenting and evolving
5. not reinventing the wheel

7.1.1 Objectives and Expectations

Assuming that you must have an SoS-like system that performs in some defined way, and that this system will require a large development effort, there are some things we now know should be done. They are to start by defining precisely what you need and why; to restrict the program's commitments to what you know how to do; and to not guess. Keep the objective definition as simple and concise as possible, preferably to only a couple of sentences. For a battlefield-management system, for example, you might write: Obtain a system that keeps every friendly warrior informed of the status of all friendly and unfriendly forces and that permits the friendly forces to cooperate in jointly achieving their objectives.

With such a simple statement, the next need is to define expectations. Here, the issue is more difficult and requires acceptance of several painful but historically obvious truths.

1. No one now knows precisely how such an SoS-like system would ultimately work or how to build it.
2. Attempting to specify and contract for a massive system with unknown capabilities and with a traditional requirements-driven process will almost certainly result in failure.
3. To avoid failure, large systems programs must have a unifying conceptual design.
4. Unifying conceptual designs are produced by knowledgeable people who truly understand both the system's operational environment and the available technology.
5. To truly understand the system's operational environment, one must have been immersed in that environment, and development groups rarely have the required operational knowledge or experience.

There are four reasons why a traditional requirements-driven contracting process would likely fail.

1. When contracting for largely unknown system capabilities and technologies, the designers generally adopt a brute-force design strategy.
2. Brute-force designs usually result in unnecessarily complex, inflexible, and expensive products.
3. As the system's requirements became better understood, development costs and schedules invariably escalate.
4. Ballooning estimates often result in complete system cancellation, and this often happens before enough is learned to establish a realistic and soundly designed follow-on program.

This history suggests why it is critically important to establish a realistic and achievable set of initial expectations and to maintain a clear focus on the essential system functionality. This in turn requires that the entire program follow a discovery-based rather than a contract-based development process. If this discovery-based process is consistently focused on the essential system characteristics, it can limit the customer's or the public's initial expectations as to what the developers know how to do. A discovery-based process has the following steps.

1. Define the core system concepts.
2. Launch multiple parallel development efforts.
    a. Keep these efforts consistent with the core concepts.
    b. Build and field early products.
3. Experiment with and evolve these early products to learn what works and why.
    a. Use the resulting knowledge to launch additional parallel development efforts.
    b. With increasing knowledge, gradually increase the amount of parallel development work.
    c. Recognize that a well-architected, suitably adaptable, and properly implemented system will keep evolving and probably never be finished.

7.1.2 Core System Concepts

Before building or contracting to build any significant part of the system, establish the system's core architectural concepts. Some of the key questions to answer with the architectural effort are the following; others will occur to you as you go along.

1. Will all of the system nodes be equal or will there be a hierarchy?
2. If there is a node hierarchy, how should the levels differ?
3. What level of dependency is required among the system's nodes?
4. How is security to be handled?
5. What are the initial node interface specifications?
6. Can these specifications be frozen now or is more information needed before doing so?

The critical need is to define the core concepts first and then to use these concepts in the specifications for all subsequent work. It is also essential to recognize that the initial interface definition will almost certainly change and to not get too heavily committed to a particular design before you know enough to have a reasonable chance of producing a flexible, adaptable, and evolvable system.

7.1.3 Parallel Development

Define a very limited set of initial node-development efforts that can be completed relatively quickly and that will be sufficient to demonstrate the system's basic emergent properties. Wherever possible, build these initial nodes from existing systems, COTS products, or even existing SoS-like systems. Don't get committed to final node products until you understand the system's basic behavior and can tune and adjust the architecture to adequately meet your current needs and to provide flexibility and expandability for the future. Since this will almost certainly involve changes to the core system architecture, limit the initial parallel development efforts to the absolute minimum needed to learn about the system's emergent behavior. Then, as knowledge grows, expand development scope, always restricting program commitments to what you know how to do.

While this might initially seem like a reasonable strategy, there is a problem. All development work is new and no one really knows in advance how any project will turn out. The key here is engineering judgment. There is a big difference between developing a brand-new unprecedented product and developing a new and challenging version of a product of a type you or someone else has previously developed. The need with SoS-like development programs is to limit the unprecedented work as much as possible and, where such efforts must be tackled, to start with prototypes and experimentation. Also, use the best known practices with both the traditional and the unprecedented work.

With such a seemingly open-ended strategy, the system's customers will almost certainly argue that the system's intended capabilities are needed as quickly as possible and that they can't wait. However, crash efforts to develop large and unprecedented systems almost always

fail. The incremental strategy is not only the fastest way to build such systems; it is likely to be the only successful way.

7.1.4 Experimenting and Evolving

Other than to reduce program risk, the next most important advantage of an incremental development strategy concerns using the latest technologies. With the rapid pace of modern technology and the speed with which new concepts and methods are introduced, it is hard to keep systems current and competitive. This is a particularly critical concern for military and public infrastructure systems where unfriendly adversaries will seek access to the systems. Such programs are necessarily involved in a constant measure and countermeasure competition and any frozen and static system will be seriously exposed. This means that the design and evolution of many such SoS-like systems must be continuous. This further implies that some overall system architectural, strategic, and standards resource must be retained to monitor the learning and evolution process, to ensure that the benefits of new knowledge are regularly incorporated into the system, and to react to newly discovered threats.

7.1.5 Not Reinventing the Wheel

A defined process is a learning vehicle: it encapsulates known best practices in an easily used and available form. As pointed out in Section 6 of this report, the use of a well-designed and properly implemented process is essential when developing or working on SoS-like systems. In developing any large-scale unprecedented system, there is no reason to struggle with previously solved problems like project planning, project management, and quality control. There are known and proven methods for addressing these problems that are now in widespread use [Davis 03, Humphrey 02].

While the massive scale of any SoS-like development effort would almost certainly cause unprecedented development problems, the incremental strategy should limit the initial program scope and permit the establishment of an overall system architecture and structure. This would isolate individual node-development issues and permit the overall program to proceed even when individual node-development efforts had serious problems. With this incremental strategy, large-scale SoS-like programs should have substantially lower risk than traditional requirements-based programs of comparable scale.

7.1.6 Strategic Considerations

While there are many ways to design and develop massive SoS-like systems, there are some key strategic considerations that involve architectural definition, program initiation, architectural monitoring and control, and development monitoring and control. Since many of the properties of these systems emerge over time, however, it is easy for systems designers to lose sight of the system’s essential architectural properties and to fail to suitably control and maintain these essential properties.

7.1.6.1 Architectural Definition

To produce the initial core system concepts and architecture, a small band of knowledgeable system architects and designers should be assembled and asked to produce the SoS design concepts. If this group has the wisdom to decide what SoS properties to specify, which ones to guide, and which ones to leave open and unconstrained, the system program will likely have a suitable design foundation. A limited number of people should be involved in this earliest program-initiation phase and the effort must follow sound architectural principles [Abowd 96, Barbacci 03, Bergey 97, Maier 98]. The suggested approach is to form a core program team and to have it produce an overall process and program plan. This planning effort should be kicked off with a standard program launch [Humphrey 06b].

7.1.6.2 Program Initiation

Planning for program initiation should be guided by a systems-initiation process that provides for an architectural and structural plan, a scale-up plan, and an overall development plan. These plans should be closely coupled with the acquisition strategy so that contractor participation in the planning effort can be obtained as soon as the contractors are selected. Key contractor-selection criteria should be the level and type of support committed to this program-initiation work. The architectural and structural plan would focus on the plans to address many of the topics discussed in Sections 1-4 of this report, and the scale-up plan would establish the program's scale-up strategy.

In this planning, the initial focus should be on leadership, team membership, and structure and on the coordination and guidance for the role-manager teams.[1] The overall development plan would be a traditional systems-development planning effort, but it would be heavily supported by and possibly largely staffed by development group representatives. As the program scales up, the structure, coaching, and operation of the leadership and role-manager teams is an important process area that needs further research. While the general concepts have been developed and proven with the SEI TSP project, there is as yet limited experience with the use of these methods with large and truly massive programs [Humphrey 06b].

7.1.6.3 Architectural Monitoring and Control

Since the architecture for an SoS is embodied in the node interfaces, and since internal node architecture and design should be opaque to the rest of the SoS, strict adherence to the architectural standards is essential. It is therefore necessary to maintain a knowledgeable and informed central resource to ensure architectural adherence, to monitor the continuing suitability of the architecture, and to evolve and maintain a strategic vision of SoS needs and architectural enhancements. This latter capability is particularly important since many of the SoS's most powerful properties will likely emerge after an initial foundation system has been developed and used.

[1] The terms *leadership team* and *role-manager team* are defined in Humphrey 06b.

7.1.6.4 Development Monitoring and Control

The monitoring and control of node development is essentially an acquisition process, although potentially a very large one. To the extent that an SoS is to meet a governmental or large-scale industrial requirement, it would almost certainly have a central systems-engineering and systems-management function to handle SoS program initiation and to support and oversee SoS architectural monitoring and control. The central systems-engineering effort could be maintained by the government agency or industrial organization itself or it could be contracted out, much as the U.S. Army has done with the FCS program.

8 Large-Scale Single-System Process Considerations

It is important to consider the process issues of larger scale node development, both because node development and management will be important for creating SoS-like systems and because prior experiences with such large-scale work should provide helpful guidance for SoS-related programs. In an SoS program, the node-development projects could range from small one- to two-person jobs to enormous programs involving several hundreds or even thousands of people. Since the processes and methods for developing small-scale systems involving fewer than several dozen people are well known, the scale-up problem of interest concerns processes for the truly massive system nodes. Examples of such systems could be computer operating systems, entire business ERP systems, or command and control situational awareness capabilities for complete integrated weapons systems.

While the particular scale-up strategy could be largely independent of a particular process, the process used must meet the aforementioned scalability criteria. As a consequence, the following discussion uses the TSP as the sample process. However, as noted above, other processes could be used if they also were shown to be scalable. In addressing the scale-up challenge, there are three likely strategies.

1. Use the selected process for multiple small-scale elements of the larger scale program to build credibility and exposure before attempting to tackle the truly massive aspects of the program.
2. Launch the process when the large-scale system is initially being architected and planned and before it grows to a massive scale.
3. Introduce the process with a large-scale system after it has been underway for some time and after it involved hundreds to thousands of people and multiple organizations with facilities in several geographical locations.

While strategies 1 and 2 are likely to be the most practical ways for introducing a new process into a massive development effort, both strategies defer the process scale-up issues to subsequent stages. For this reason, the following discussion describes and addresses the problems of introducing the TSP into a large-scale systems program that has been underway for some time and that involves many hundreds or thousands of people in multiple organizations and locations.

A critical task for any process-introduction program concerns providing adequate training and process experience. Since the TSP requires that team members and their management be specially trained and since it calls for a suitable number of qualified TSP coaches to be available

to launch and support both the software-development and systems-engineering teams, most initial large-scale process-introduction programs are gradual by necessity. However, since this need would again defer discussion of the real process scale-up problems that are the subject of this report, it is here assumed that the organizations involved already have a TSP-trained staff, that the development teams and their management already have TSP experience, and that suitable numbers of qualified and experienced TSP coaches are available.

While these would be unrealistic assumptions for any current large-scale program candidate, solutions for these skill-building and experience problems are well known [Humphrey 02]. Also, since the methods for introducing and using the TSP with moderate-sized programs are described and available, they are not addressed here [Humphrey 02, 06b]. However, strategies and methods for handling the five remaining scale-up issues have not been developed or demonstrated. These five issues are as follows.

1. structure and operation of the leadership team
2. structure, leadership, and guidance of the role-manager teams
3. methods and practices used to audit and ensure process compliance
4. methods for tracking and reporting on team and program status
5. methods for aligning goals and objectives for multiple autonomous constituencies

Where this and the following discussion refer to aspects of the TSP that are not familiar, the reader should consult the appropriate TSP publications [Humphrey 02, 05, 06a, 06b].

8.1 The Leadership Team

When the TSP multi-team process (TSPm) is used to develop a large-scale single system, the leadership team is composed of the overall program manager and the leaders of the development and other project teams. This structure, however, is designed for projects with only a single level of team management and cases where the number of project teams is small enough to permit all of the team leaders to be members of a reasonably sized leadership team.

Experience with the TSP has shown that teams with more than about 10 to 12 members generally do not have the cohesion and cooperation required for truly creative, effective, and productive teams. As a consequence, this single-level structure is suitable only for projects with up to about 150 to 200 team members. Since truly massive systems-development programs will be much larger than this, a process to support any such program must provide a means for structuring leadership teams with up to hundreds of teams, dozens of team leaders, and a many-layered or network-like management structure. While the particular structure selected for each program will almost certainly be unique to that program, the various options and the issues to consider in selecting a specific approach are as follows.

1. Decentralization is the most common method used by large corporations to handle the span-of-control problem. Here, the approach is to identify a number of responsible subordinate-level executives and to give them sufficiently defined responsibilities to run their own groups as complete businesses. There are two common ways to do this. In the

first, holding companies follow the SoS strategy. However, they avoid the scale-up issues being considered in this section; for that reason, this approach is not considered further. The second, the divisional model, has a corporate entity that exercises overall command but delegates control to division management. Except for resolving interdivisional issues and monitoring goals and business performance, the rest of the work is delegated. When large SoS-like systems are needed, this is likely to be the preferred management model.

2. A second approach would be to establish a hierarchical structure with multiple levels of leadership teams. That is, at the lowest leadership level, several intermediate-level program managers would each chair a leadership team with multiple team leaders as members. At the next level, higher level assistant program managers would chair leadership teams with multiple intermediate-level program managers, and so forth. If the average span of control of these teams were each 5 to 10, then one layer could handle programs with 25 to 100 or so developers, two layers could handle several hundred developers, and three layers could handle a few thousand. While this does not appear to be an excessively deep management structure, at least based on the structure of many current-day corporations, it imposes additional management layers, slows decisions, and narrows the communications bandwidth between the most senior program management level and the working-level team members.
3. A third common organizational strategy is to broaden the management span of control to dozens or more teams and team leaders, and to provide these management levels with staff support to help oversee and manage their expanded groups. If the working-level teams still had an average of 10 members each, but the leadership teams averaged 20 members, a single layer structure could handle up to 200 or so team members, and two levels could handle nearly 4,000.

While alternative 3 would seem the most attractive approach, it substantially dilutes the cohesion and responsiveness of the leadership teams and introduces an intermediate staff layer that could inhibit vertical and horizontal communication. The management problems of alternative 2 would remain, but they would be somewhat reduced and replaced by the introduction of multiple staff members. Staff support groups can be very helpful, but they do not have management responsibility and tend to act parochially. This typically results from their more specialized focus and their common tendency to view their personal areas of responsibility as primary. This can favor one specialty area over another and reinforce the stovepipe tendencies common in large organizations. This would reduce the level of broad-gauge executive attention available to resolve issues that are escalated from conflicting groups at lower levels.

Since there aren’t any other proven effective ways to handle this management span-of-control problem, some combination of these three approaches must be selected for each case. It would also be possible to use a mixed strategy with some parts of the management hierarchy following one approach while others followed a different one. Also, the TSP or any other process that followed these principles should rely on self-directed teams. Since TSP teams

typically place reduced demands on their management, spans-of-control could be somewhat greater than with other groups. The most effective strategy in each case would depend on the personal skills and preferences of the people involved and the degree to which common benefits and cross-platform responsibilities can be established. There is no universally best approach for all cases.

8.2 The Role-Manager Teams

In the TSP, the team members participate in team management. To do this, the TSP team-building process guides teams through a launch in which they define their development processes and produce their project plans [Humphrey 06b]. During the launch, the team also agrees on the roles each member will perform to support the team. There are four standard TSP roles for process support.

1. planning manager
2. process manager
3. quality manager
4. support manager

There are also four roles for product-related topics.

1. customer-interface (requirements) manager
2. design manager
3. implementation manager
4. test manager

Additional roles can be added if needed. Typical additions are safety manager, security manager, subcontract manager, and communications manager.

The TSP roles help teams to manage their projects by designating members as responsible for various management topics. For example, if one member encountered problems in obtaining timely test facilities, he or she could raise the issue with the test manager, who would look into the matter and see that it was promptly resolved. While the test manager would not necessarily do all of the testing or resolve all of the test-related issues, he or she would be responsible for ensuring that these issues were resolved and that all of the test-related work was properly completed. As a result, TSP teams can generally identify and resolve most risks and issues early enough to avoid the time-consuming late surprises that often delay other development groups.

When scaling the TSP up to larger projects with the TSPm process, the role managers are grouped into role-manager teams, as shown in Figure 1. Here all of the planning managers from all of the teams in a large project work together as a planning-manager team to resolve

cross-team planning issues, often without having to involve the leadership team. These role-manager teams also serve as specialized resources that the leadership team uses for help and guidance on cross-team issues. For example, the planning manager team could identify critical cross-team schedule dependencies, or the design-manager team could recommend how to handle an interface issue. The role-manager teams also provide the team members with a rapid communication link across the entire program.

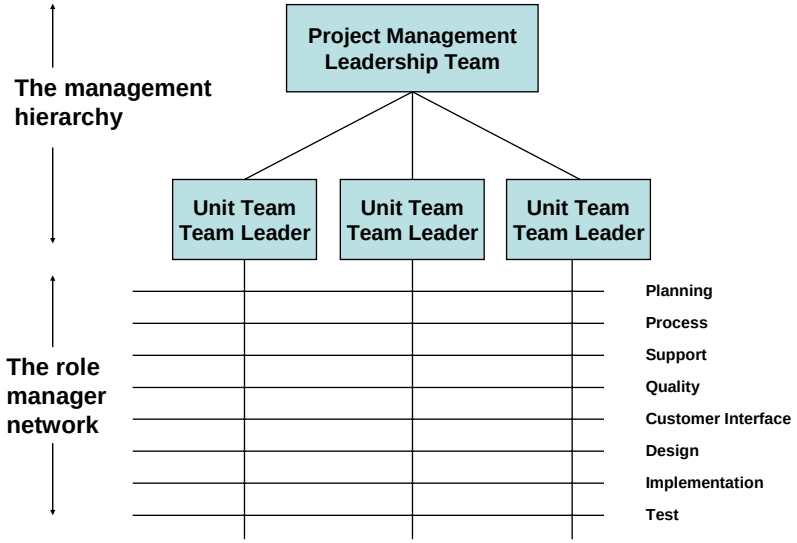

Figure 1: Multi-Team Role-Manager Communication Network

As program size grows, choices must be made about how to structure and relate the role-manager teams. Here, the choices are much the same as for structuring the leadership team: establish multiple hierarchical layers or expand the span of control with staff support. Larger spans of control would be more practical for role-manager teams than for management teams, and staff coordination and support would probably be more appropriate and useful. However, the specific role-manager-team structure selected must vary depending on the overall program structure, the personalities involved, and the issues to be addressed. It might also be appropriate to establish different role-manager team structures for different role-manager teams.

8.3 Process Compliance

Process compliance is a particularly important issue when scaling up programs because the greatly increased product size makes the consequences of poor-quality work much more damaging. It is for this reason that large-scale systems programs must pay particular attention to the quality of the processes adopted by their various development groups and that they es-

tablish means to ensure that all of the work on all component parts of the system is done according to these established processes.

The next question is: How can one determine if all of the work is actually being done in accordance with the established processes? There are only two ways to do this: (1) to assess the quality of the product after it is produced and (2) to assess the quality of the process as it is being enacted. While any massive system would almost certainly be built in multiple cyclic steps, the first step would usually be the largest and most critical. Since it would be unwise to have these first critical parts be of poor quality, it is essential that process quality be monitored and assured during its enactment, and that any process deviations be promptly corrected.

To do this, however, the process must be measured, and these measures must be sufficiently detailed, accurate, precise, and complete to permit a comprehensive process analysis and to pinpoint areas where process enactment is deficient and in need of adjustment. This, in turn, leads to the final topic of this paper: program tracking and status reporting.

8.4 Program Tracking and Status Reporting

The most fundamental question in this entire report is this: With the truly massive systems programs of the future, how can one possibly measure status and track progress? This raises the question discussed in Section 4 on measurement scalability. For measures to be scalable, they must be derived from complete, accurate, precise, and auditable measures, not from after-the-fact guesses and approximations. Since traditional time-card, expense report, and test-defect data are generally obtained after the fact, the data are rarely complete, accurate, or precise.

For development work, only three basic measures are needed: the time spent on each project task, the sizes of all of the products produced, and data on all of the defects found. However, all of these data must be obtained for every task of every process phase and for every component of the entire system. While this is usually an enormous amount of data, experience shows that data-gathering only takes a few percent of a large program's effort. However, having such data provides a complete and precise picture of the status of every part of the program, of the quality of the processes enacted by every development group, and of the quality of all of the components and parts produced throughout the entire program. While having such data is enormously valuable, there are two key questions: (1) How do you get such data? and (2) How should you use such data?

8.4.1 Getting Accurate Process Data

Since the cost and time spent in systems-engineering and software-development work consists almost entirely of the time and cost of employing and supporting the systems engineers and software developers, and since what they do is almost entirely knowledge work, the required measures must relate to what these professionals are thinking and doing. While some

of this work is often observable and can be independently tracked and measured, such measures are necessarily incomplete, inexact, and imprecise. Therefore, for truly usable process data, the professionals must gather these data themselves. However, for knowledge workers to consistently gather complete, accurate, and precise data on their own work, four conditions must be met.

1. The accuracy, completeness and precision of these data must be both relevant and important to the professionals.
2. The professionals must be willing and able to gather the data.
3. The professionals must actually use the data in their own work.
4. The data must not be perceived as threatening.

While the first two points are self-evident, the last two are not. The reason that the professionals must use the data is that, if they do not, they will not be motivated to gather complete and accurate data and to keep doing so even under extreme schedule pressure. This is critical, for professionals will invariably come under intense pressure to finish their project on time, and this is usually when their projects are late and striving to catch up. This is precisely when it is most important that they have the data needed to accurately determine project status. The fourth condition, that the data not be perceived as threatening, is addressed in the next section on using process data.

8.4.2 Using Accurate Process Data

The first rule in using process data is to ensure that these data are not perceived as threatening to the people who gather the data. If, for any reason, the systems engineers or software developers suspect, even wrongly, that their data is being used by management to evaluate their personal work, they will either stop gathering the data, or they will only provide data that show the kind of performance they believe management wants. So if any manager insists that he or she must have personal data on any team member's work, the news will immediately spread throughout the organization and no manager anywhere on the entire program will get complete, accurate, or reliable data thereafter.

While some managers will argue that, without such data, they cannot evaluate their people, this is nonsense. First, these managers never had such data before and they were presumably able to make proper evaluations. Second, if the managers suspected some team members of being incompetent, these members' data would likely have been incompetently gathered and thus useless.

Ensuring that the required process data are not perceived as threatening is relatively simple. It only requires that management never ask to see or use any individual professional's personal data for any reason. As long as the development teams are reasonably sized, then composite team data can be used for all process and project analysis purposes.

It is not easy to meet all of the conditions for consistent and complete data gathering. It requires training, coaching, tool support, and continued management priority. But, as experience with the TSP has demonstrated, once the proper conditions are met, and as long as these conditions continue to be met, development groups will gather and continue gathering and using the required data [Humphrey 02, 05].

Once management has access to detailed team level process data, the next question concerns using that data. The TSP process has defined a family of measures that teams can use to report their status to management. While these measures are useful and effective at the team level, there are many ways to report these data to higher management levels. The proper ways to do this for truly massive development programs have not yet been developed and remain an important task to be completed as part of any early process and program-management studies and trials with large-scale SoS-like development programs.

8.4.3 Aligning Goals and Objectives

Convincing independent groups to cooperate is one of the most challenging concerns of modern society. If there were a magic answer, it would already have been applied in the political, diplomatic, and military spheres. However, there are known and proven methods for obtaining the cooperation of independent groups in the more constrained environment of system development. They involve the teambuilding and coaching practices that have been embodied in the TSP [Humphrey 06b].

These teambuilding methods have been well demonstrated with systems of a few dozen to a hundred or more developers, but there are no published data on their use with truly massive systems programs. While there is no known reason why the methods would not work, it is not obvious how to scale them up or how to introduce them. This is one of the principal areas where additional research and experimentation is required before we can confidently embark on massive SoS-like systems programs.

9 Conclusions

This report concludes that, to successfully produce the large complex, and critical systems of the future, the development processes must meet the following criteria.

- predictably control development cost and schedule
- responsively handle changing needs
- minimize the development schedule
- be scalable
- predictably produce quality products

These conclusions were reached through the following logic.

- Large and complex computer-based systems are now critical to the economic and military well-being of the United States and to much of the developed world.
- With current development methods, implementation of these systems is typically late, over cost, and results in poor-quality products.
- Since future development programs will be far more challenging than those of today, continuing to use the methods of the past will produce even worse results, and often will not deliver any products at all.
- To properly address these problems, organizations must adopt sound processes that can be scaled up to the larger challenges of the future.
- For processes to be scalable, they must be based on well-defined methods, precise measures, and statistically managed quality.
- For these processes to be properly and consistently used, everyone at every level in the organization must use them.

This logic is based on the following facts.

- Large-scale systems-development efforts have typically failed because of management and not for technical reasons.
- While known and effective solutions to these problems exist, they are not widely practiced.

- Poor management practices damage technical programs by increasing their costs, delaying their completion, and masking critical technical problems.
- By applying known and proven management and quality methods, organizations can improve the cost, schedule, timeliness, and quality of their work.
- If these known and understood project-management methods are not properly applied, most large-scale system-development programs will be unmanageable.
- Introducing proper methods and practices takes substantial effort and requires continued management support.
- With management support, systems-engineering and software-development professionals can be trained to use proper methods and to manage their own work, improving the organization's ability to handle large-scale work.
- With proper training and coaching, teams of trained professionals will follow effective processes and use sound methods.
- When all of the professionals on all of the development teams consistently use sound methods, they produce superior results.

The final conclusion is that the current commonly used systems-development methods have reached (or soon will reach) their feasibility limits, and continuing to develop the increasingly challenging and massive systems of the future with the outdated methods of the past is destined to failure. Furthermore, continuing to address the technology issues of SoS-like systems without simultaneously addressing their process-management, team-management, and quality-management issues will be counter-productive and could even be dangerous. The danger is that, with the rapid pace of technology, society could well be lulled into the false belief that the technical community is capable of building the systems we can technically describe. As these newer systems are used to support increasingly critical aspects of modern society, we then will likely face far more catastrophic system failures than we have experienced heretofore.

In summary, this report recommends that efforts be established, staffed, and funded to refine, test, and widely adopt processes that meet the requirements outlined in this report and that early projects be identified for testing and refining these processes on large-scale systems of the type now being planned by the U.S. Department of Defense and other organizations.

References/Bibliography

URLs are valid as of the publication date of this document.

[Abowd 96] Abowd, Gregory; Bass, Len; Clements, Paul; Kazman, Rick; Northrop, Linda & Zarenski, Amy. *Recommended Best Industrial Practice for Software Architecture Evaluation* (CMU/SEI-96-TR-025, ADA320786). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. http://www.sei.cmu.edu /publications/documents/96.reports/96.tr.025.html.

[Albert 02] Albert, Cecilia & Brownsword, Lisa. In collaboration with Bentley, David; Bono, Thomas; Morris, Edwin & Pruitt, Deborah. *Evolutionary Process for Integrating COTS-Based Systems (EPIC*) (CMU/SEI-2002-TR-005, ADA408653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports /02tr005.html.

[Barbacci 03] Barbacci, Mario; Clements, Paul; Lattanze, Anthony; Northrop, Linda & Wood, William. *Using the Architecture Tradeoff Analysis Method (ATM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study* (CMU/SEI-2003-TN-012, ADA418415). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu /publications/documents/03.reports /03tn012.html.

[Bergey 97] Bergey, J. K.; Northrop, L. M.; & Smith, D. B. *Enterprise Framework for the Disciplined Evolution of Legacy Systems* (CMU/SEI-97-TR-007, ADA330880). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/publications /documents/97.reports /97tr007/97tr007abstract.html.

[Chrissis 03] Chrissis, Mary Beth; Konrad, Mike; & Shrum, Sandy. *CMMI – Guidelines for Process Integration and Process Improvement*. Reading, MA: Addison Wesley, 2003.

[Davis 03] Davis, N. & Mullaney, J. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014, ADA418430). Pittsburgh, PA: Software Engineering Institute, Car-

negie Mellon University, 2003. http://www.sei.cmu.edu /publications/documents/03.reports /03tr014.html.

[Deming 00] Deming, W. Edwards. *The New Economics for Industry, Government, Education, 2nd Edition*. Cambridge, MA: The MIT Press, 2000.

[Fisher 06] Fisher, David A. *An Emergent Perspective on Interoperation in Systems of Systems* (CMU/SEI-2006-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006. http://www.sei.cmu.edu/publications/documents/06.reports /06tr003.html.

[GAO 99] General Accounting Office (GAO). *Observations on FAA's Air Traffic Control Modernization Program*. http://www.gao.gov/archive/1999/r199137t.pdf.

[Grojeans 05] Grojeans, Carol A. "Microsoft's IT Organization Uses PSP/TSP to Achieve Engineering Excellence." *CrossTalk*, March 2005. http://www.stsc.hill.af.mil/Crosstalk/2005/03/0503Grojean.html.

[Gross 05] Gross, Grant. "FBI Trying To Salvage $170M Software Package." *Computerworld Business Intelligence*, January 14, 2005, http://www.computerworld.com/databasetopics/data/story /0,10801,98980,00.html.

[Hansen 99] Hansen, Wilfred J. *Construction and Deployment Scripts for COTS-Based, Open Source Systems* (CMU/SEI-99-TR-0013, ADA373330). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. http://www.sei.cmu.edu /publications/documents/99.reports /99tr013/99tr013abstract.html.

[Humphrey 02] Humphrey, W. S. *Winning with Software: an Executive Strategy*. Reading, MA: Addison-Wesley, 2002.

[Humphrey 05] Humphrey, W. S. PSP: *A Self-Improvement Process for Software Engineers*. Reading, MA: Addison-Wesley, 2005.

[Humphrey 06a] Humphrey, W. S. TSP: *Leading a Development Team*. Reading, MA: Addison-Wesley, 2006.

[Humphrey 06b] Humphrey, W. S. TSP: *Coaching Development Teams*. Reading, MA: Addison-Wesley, 2006.

[Juran 88] Juran, J. M. & Gryna, Frank M. *Juran's Quality Control Handbook, 4th Edition*. New York, NY: McGraw-Hill Book Company, 1988.

[Kay 03] Kay, Russel. “Buffer Overflow.” *Computerworld Security*, July 14, 2003. http://www.computerworld.com/securitytopics/security /story /0,10801,82920,00.html.

[Lewis 04] Lewis, Grace A.; Morris, Edwin J.; & Wrage, Lutz. *Promising Technologies for Future Systems* (CMU/SEI-2004-TN-043, ADA431162). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu /publications /documents/04.reports/04tn043.html.

[Maier 98] Maier, Mark W. “Architecting Principles for Systems-of-Systems.” *Systems Engineering 1*, 4 (1998): 267-284.

[Maldonado 06] Maldonado, Jose C.; Carver, Jeffrey; Shull, Forrest; Fabbri, Sandra; Doria, Emerson; Martimiano, Luciana; Mendonca, Manoel; & Basili, Victor. “Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness.” *Empirical Software Engineering 11*, 1 (March 2006): 119-142.

[McAndrews 00] McAndrews, Donald R. *The Team Software Process (TSP): An Overview and Preliminary Results of Using Disciplined Practices* (CMU/SEI-2000-TR-015, ADA387260). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/publications/documents/00.reports /00tr015.html.

[Northrop 06] Northrop, Linda; Feiler, Peter; Gabriel, Richard P.; Goodenough, John; Linger, Rick; Longstaff, Tom; Kazman, Rick; Klein, Mark; Schmidt, Douglas; Sullivan, Kevin; & Wallnau, Kurt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[Paulk 95] Paulk, Mark C.; Weber, Charles V.; Curtis, Bill; & Chrissis, Mary Beth. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley, 1995.

[Pracchia 04] Pracchia, Lisa. “The AV-8B Team Learns Synergy of EVM and TSP Accelerates Software Process Improvement.” *CrossTalk*, January 2004. http://www.stsc.hill.af.mil/Crosstalk/2004/01 /0401Pracchia.html.

[Rickets 05] Rickets, Chris A. “A TSP Software Maintenance Life Cycle.” *CrossTalk*, March 2005. http://www.stsc.hill.af.mil/Crosstalk /2005/03/0503Rickets.html.

[Sai 04] Sai, Vijay. *COTS Acquisition Evaluation Process: Preacher's Practice* (CMU/SEI-2004-TN-001, ADA421675). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn001.html.

[Seacord 06] Seacord, Robert C. *Secure Coding in C and C++*. Reading, MA: Addison-Wesley, 2006.

[Trechter 05] Trechter, Ray & Hirmanpour, Iraj. "Experiences with TSP Technology Insertion." *CrossTalk*, March 2005. http://www.stsc.hill.af.mil/Crosstalk/2005/03/0503Trechter.html.

[Zachary 94] Zachary. G. Pascal. *Showstopper!* New York, NY: The Free Press, 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2006	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Systems of Systems: Scaling Up the Development Process	FA8721-05-C-0003

6. AUTHOR(S)

Watts Humphrey

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2006-TR-017

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2006-017

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Some systems have some but not all properties of a system of systems (SoS). We refer to these as SoS-like systems. This report reviews the fundamental process and project-management problems of large-scale SoS-like programs and outlines steps to address these problems. The report has eight sections. Section 1 summarizes current thinking on the nature of future complex systems, and Section 2 discusses the systems-design problems of the future, particularly the partitioning of massive systems into system-of-systems structures. Section 3 points out how large-scale systems development efforts have typically failed because of project-management and not technical problems, and that the solutions to these problems are known and highly effective, but not widely practiced. It explains why, if the project-management problems of the past are not promptly and effectively addressed, large-scale systems development programs will likely be unmanageable. Section 4 discusses the requirements for a scalable process, and Section 5 both reviews and explains the quality-management principles upon which any scalable process must rest. Section 6 reviews the nature of the project-management problems currently faced by large-scale software-intensive system development efforts and explains why attempts to scale up current methods to very large-scale systems work will almost certainly fail. Section 7 describes process strategies for supporting development of a network-like system of systems and it outlines the process and project-management topics needing further research and development. Finally, Section 8 reviews the process considerations for supporting the very large-scale integrated development programs of the future. The report concludes that, unless steps like those outlined in this report are taken in conjunction with continuing technical research and development, the large-scale systems development efforts of the future will almost certainly fail, and often catastrophically.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Systems of systems, SoS, scalability, process strategy, process management	70

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL